Examining ASP.NET's Membership, Roles, and Profile - Part 7

By Scott Mitchell

Introduction

One of the main challenges of building a programming framework is balancing the desires to create a standardized, straightforward API for accomplishing common tasks and providing flexibility and customizability so that developers using the framework can mold the framework to fit their applications rather than having to bend their applications to fit the framework. With the .NET Framework 2.0, Microsoft implemented this delicate balance with the *provider model*. The provider model allows for a standardized API to be defined within the framework, but enables developers to design their own concrete implementation and plug that implementation into their systems at runtime. (See A Look at ASP.NET 2.0's Provider Model for more details on the provider model, its implementation in ASP.NET 2.0, and the concepts behind it.)

The Membership, Roles, and Provider pieces examined throughout this article series all utilize the provider model. Throughout the past six installments of this article series we've examined some of the providers that ship with the .NET Framework (such as `SqlMembershipProvider`, `SqlRoleProvider`, and `SqlProfileProvider`). In fact, you can download the source code for these built-in providers from Microsoft as well as a Provider Toolkit for creating your own custom providers [link]. And Scott Guthrie lists a number of custom Membership, Roles, and Profile providers in his ASP.NET 2.0 Membership, Roles, Forms Authentication, and Security Resources page.

If the none of the built-in providers meet your needs, you can always create your own custom providers. Custom providers are useful when integrating legacy systems (perhaps you have an old `Users` database table that differs from the database tables used by `SqlMembershipProvider`) or when needing to persist membership, role, or provider information in a backing store not natively supported by one of the built-in providers.

In this article, we'll discuss the concepts behind creating and using a custom provider and then implement a simple custom profile provider that serializes profile information to XML files. Read on to learn more!

The Responsibility of the Profile System

As discussed in Part 6 of this article series, the Profile system is responsible for storing additional user-specific information. The Profile system allows the page developer to define a set of "profile properties" in the `Web.config` file, and then needs to save and read those property values from some backing store upon request. The .NET Framework 2.0 ships with a `SqlProfileProvider`, which persists these user-specific properties to a denormalized database table (namely, it squishes all of the profile property names and values into single columns in a database table). Microsoft also offers the custom Table Profile Provider, which uses a normalized data model to store the Profile properties, using a separate column in a database table for each property.

If neither of these Profile providers meets your needs, you can always create your own custom provider. Creating and using a custom provider - whether it be for Profile, Membership, Roles, the ASP.NET site map, or some other subsystem - entails the following steps:

1. **Create the custom provider** - this involves creating a class that derives from the appropriate base class. To extend the Profile system, the provider must derive from the `ProfileProvider` class.
2. **Configure the application to use the custom provider** - we need to configure the application's `Web.config` file to specify that the Profile system should use our custom provider in lieu of the default one (`SqlProfileProvider`).

Microsoft's Provider Toolkit web page provides a great starting point for learning about and creating a custom provider. For the Profile system, there are two documents worth reading: Introduction to Profile Providers, which provides a conceptual overview of the Profile system and the responsibility of its providers, and ASP.NET 2.0's Profile Providers, which offers a more concrete look at how the `SqlProfileProvider` in ASP.NET 2.0 is implemented.

Recently I was discussing the default Profile provider (`SqlProfileProvider`) with a colleague and he inquired as to whether there was a custom Profile provider that persisted data in an XML file format. I subsequently Googled for any results, but didn't find anything and therefore decided to try my hand at creating such a custom provider. The remainder of this article examines my implementation of `XmlProfileProvider`, a custom Profile provider that persists user-specific settings in XML files in the `App_Data` folder. The complete code (along with a simple test web application) can be downloaded at the end of this article.

Designing the `XmlProfileProvider`

The default SQL-based providers for the Membership, Roles, and Profile systems are all designed to store information for multiple applications. This is evidenced by the `aspnet_Applications` table, which has a record for each application instance; each user in the system is associated with an application via the `ApplicationId` foreign key in the `aspnet_Users` table. I decided to replicate this functionality for my profile provider (since, conceivably, the `XmlProfileProvider` might be used in conjunction with the SQL-based Membership and Roles providers). The Profile data is serialized (by default) to files in the `App_Data` folder so as to prevent users from directly requesting these files through a browser. (If a user attempts to visit `http://www.yoursite.com/App_Data/`*`anyFile`* through their web browser, ASP.NET will return an HTTP 404 message.) You can, however, customize the base folder the XML files are saved under via the custom provider's `profileFolder` setting.

Specifically, the Profile data is stored in the following format: `~/App_Data/XmlProfiles/`*`applicationName`*`/`*`username`*`.xml`. Each XML file starts with a `<profileData>` root element, but its children elements are determined by the Profile properties spelled out in `Web.config`. As discussed in Part 6 of this article series, the Profile properties can be serialized as strings, XML, or binary data. For string-serialized properties, the XML file saves the result as text content. For XML-serialized properties, the raw XML is saved, and for binary-serialized content, the base-64 encoded binary content is persisted.

To illustrate the file naming and XML content persistence, imagine that our web application used six Profile properties defined using the following `<properties>` definition in `Web.config`:

```
<profile defaultProvider="...">
   <providers>
      ...
   </providers>

   <properties>
      <add name="Birthdate" type="DateTime" serializeAs="String" />
      <add name="HomepageUrl" type="String"/>
      <add name="Signature" />
      <add name="IncludeSignatureInPosts" type="Boolean"/>
```

```
      <add name="BillingAddress" type="Address" serializeAs="Xml" />
      <add name="ShippingAddress" type="Address" serializeAs="Binary" />
  </properties>
</profile>
```

Here, the type `Address` is defined as a simple class in the `App_Code` folder, and has scalar properties like `Address1`, `Address2`, `City`, and so on. Note that the `BillingAddress` Profile property serializes the `Address` type as XML, while `ShippingAddress` serializes it as binary data.

Given the above Profile property definition, if a user with username "Jisun" were to interact with the Profile system when visiting a page on our website whose application name was configured to "MyApp", a file named `~/App_Data/XmlProfiles/MyApp/Jisun.xml` would be created, and might contain the following markup:

```
<?xml version="1.0" encoding="utf-8"?>
<profileData>
  <IncludeSignatureInPosts>False</IncludeSignatureInPosts>
  <Signature>What is life, but a seemingly random collection of 1s and 0s?</Signature>
  <HomepageUrl>http://www.datawebcontrols.com</HomepageUrl>
  <Birthdate>1979-04-01</Birthdate>
  <ShippingAddress><![CDATA[AAEAAAD/////AQAAAAAAAAAMAHBfQ29kZS5j...]]></ShippingAddress>
  <BillingAddress>
    <Address xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
      <Address1>123 Anywhere St.</Address1>
      <Address2 />
      <City>San Diego</City>
      <State>CA</State>
      <ZipCode>92109</ZipCode>
    </Address>
  </BillingAddress>
</profileData>
```

As you can see, each Profile property is persisted as an XML element as a child element of the `<profileData>` root element. For string-serialized properties - like `IncludeSignatureInPosts`, `Signature`, `HomepageUrl`, and `Birthdate` - the property value is persisted as the text value for the XML element. For the XML-serialized property (`BillingAddress`), the XML serialized content is stored in raw format as content within the `<BillingAddress>` element. The binary-serialized content of the `ShippingAddress` property is base-64 encoded.

Programming the `XmlProfileProvider` Class

When building a custom Profile provider, have it extend the `System.Web.Profile.ProfileProvider` class. At minimum, we need to provide the functionality for the `GetPropertyValues(`*`context`*`, `*`collection`*`)` and `SetPropertyValues(`*`context`*`, `*`collection`*`)` methods. Both methods accept two input parameters - *context* and *collection*. The *context* provides two bits of vital information - the username of the person whose Profile to access/save and whether the user is authenticated. (ASP.NET supports the notion of anonymous profiles. This is a topic beyond the scope of this article, however. See the Profiles section on the ASP.NET Quickstarts for more information.) The *collection* provides the set of Profile properties.

The `GetPropertyValues` method is responsible for returning the values for the specifies set of Profile properties, whereas the `SetPropertyValues`'s sole task is to write the property values specified in *collection* to the backing store (XML files, in our case). There are additional methods in the `ProfileProvider` class, but we can create a fully working custom profile provider by implementing just these two methods.

One of the challenges with creating a custom Profile provider is correctly handling the serialization and deserialization issues. The *collection* passed into the `GetPropertyValues` method indicates how each property is configured to be serialized. We can use this information to correct deserialize the data when reading it back from the XML file. For example, when working with a Profile property that has been configured to use binary serialization, we need to decode the base-64 value from a string back into the binary format (a byte array). This is done using the `Convert.FromBase64String()` method in the .NET Framework. However, if we are dealing with a Profile property that's serialized as XML or a string, we want to just read the XML content as text. On the `SetPropertyValues` side, we need to know how to serialize the data back to the XML file. For binary-serialized properties, we need to base64-encode it. XML-serialized data includes the `<?xml version="1.0" encoding="utf-8"?>` pre-processing statement, which needs to be stripped out before writing the XML data as child content for the Profile property element.

Plugging the `XmlProfileProvider` Custom Provider Into Your Website

In order to use the `XmlProfileProvider` custom provider in your web application, you first need to download the code at the end of this article. Drop the `CustomProviders.dll` assembly into your web application's `/bin` directory and then update your `Web.config` file to specify the custom provider for the Profiles subsystem:

```
<profile defaultProvider="MyCustomProfileProvider">
   <providers>
      <clear />

      <add name="MyCustomProfileProvider"
           type="CustomProviders.XmlProfileProvider, CustomProviders"
           applicationName="applicationName"
           [profileFolder="baseFolderPathForProfileXMLFiles"] />
   </providers>

   <properties>
      ...
  </properties>
</profile>
```

The `profileFolder` attribute is optional. If omitted, it defaults to `~/App_Data/XmlProfiles/`.

The download available at the end of this article includes a test web application that illustrates how to use the custom `XmlProfileProvider` class in conjunction with the default `SqlMembershipProvider` class.

Conclusion

The provider model allows for Microsoft's .NET Framework 2.0 to offer a consistent, standardized API against which Web controls, the Framework itself, and third-party libraries can be created against, but that still enables the flexibility to customize the implementation details. In this article we saw how to utilize the provider model to create a custom Profile provider, one that persists user-specific settings to XML files rather than a database.

Happy Programming!

- By Scott Mitchell

Attachments

- Download the code used in this article

Article Information	
Article Title:	ASP.NET.Examining ASP.NET's Membership, Roles, and Profile - Part 7
Article Author:	Scott Mitchell
Published Date:	October 17, 2006
Article URL:	http://www.4GuysFromRolla.com/articles/101806-1.aspx